

```
<?xml version="1.0" encoding="utf-8" ?>
```

```
<checksum>
```

```
<item keystroke="a" title="Algorithmic Writing Systems" subtitle="Douglas Edric Stanley" url="http://www.abstractmachine.net/">
```

```
<p></p>
```

```
</item>
```

```
<item keystroke="b" title="machine_language != language_machine">
```

```
<p>One of the key terms that we use to describe computer programs, is in fact one of the very terms that blinds us to the actual contours of that activity. I am speaking of the notion of a computer programming "language", and that entrance into the world of a computer program must somehow traverse a language machine. While there are indeed notions of vocabulary and syntax in many programming languages, reducing "language" to the mere mechanical stringing-together of sign-token after sign-token through some overly-fussy and arcane syntax, leaves us with an extremely approximate idea of what is entailed in "language".</p>
```

```
</item>
```

```
<item keystroke="c" title="Abstract Machines">
```

```
<p>Instead, might I suggest that we look beyond the computer as a language machine, and observe it instead from its material substrates as a mosaic of dynamic relays capable of arranging themselves into ever more recursively complex waves of abstract-able manipulations. In other words, to observe the computer as an elegant contradiction: explicitly material, implicitly immaterial, a physical machine that generates through its physicality abstract networks that render their physical nature malleable. Abstractions within a plane of immanent physicality, hence the use of a Turing terminology re-routed via Deleuze and Guattari: abstract machines.</p>
```

```
</item>
```

```
<item keystroke="d" title="Physicalisation">
```

```
<p>Elsewhere I have termed this process a process of "physicalization". Note the use of the suffix "-ization". The physical is not our point of departure, it is our product: the physical substrate is the base material -- there is no escaping the material world --, but it is also what is redesigned -- figurally -- by that same material. This "product" is a very different beast than the physical ingredients that went into it. A hyper-physicality, understood as the tendency of the machine to continually return to its underlying physical processes.</p>
```

```
</item>
```

<item keystroke="e" title="Achille's Heel">

<p>Perhaps the most obvious technical illustration today of this hyper-physicality would be the Field-Programmable Gate Array, i.e. a computer circuit that can be materially redesigned from within the circuit itself. That is its positive aspect. But physicalization is also the internal dynamic of the machine as part of a larger dialectic. From its negative we find a trauma of the material substrate that grips the algorithm from within its machinic incarnation. Ironically, the more the machine (and its users) yearn for a purer, more immaterial, in other words a more Platonic form of abstraction, the more the physicality of the machine snaps back with a vengeance. This is by design and can be traced from the very dipping of the machine into the mystical river that made it possible: it was the spatial and temporal realms that saved mathematics from its crisis and led to the invention of the machine itself. This is the strange two-step of physicalization, and the gap where artists can slip in: a machine of immanent immateriality.</p>

</item>

<item keystroke="f" title="Mosaics" image="object.png" imagetag="The Object Machine, Douglas Edric Stanley">

<p>Returning to the question of computer programming, and keeping in mind the process of physicalization -- what is actually at work inside of a computer "language" is in fact a series of operations that resemble more a patchwork, a quilt, or a puzzle, than a chain of signifiers, and in their functioning act more as switches, shunts, and regulators, than as syntagms. How do we render visible then this process; how do we move from the array() value table to the tableau? The trick in describing these program structures is how to qualify their more abstract qualities with these very basic building blocks of material. If, for example, one of the central principles of the machine is its recursivity, obviously we would need a method to somehow move from the specific shapes and colors of the individual tiles of the machine to the figure formed within the mosaic it forms. These is where the problem of representation enters into the picture, and ultimately the limits of representation in a world of hyper-forms where output generates input and is manipulated via logics of simulation rather than mimesis.</p>

</item>

<item keystroke="g" title="Terminals">

<p>How then do we program these machines? What are the devices we use to flip these switches and regulate these flows? One of the first things an artist notices when choosing his or her programming "tool", is that these do not all come in the same variety. The most obvious split in programming environments involves a choice over text-based programming or visual programming via wires, sliders and widgets. While these are indeed two important entry-level families of programming environments, they are by no means the only form of algorithmic writing, and will hopefully become less and less the standard in the future.</p>

</item>

<item keystroke="h" title="The Desktop" image="hypertable.png" imagetag="Hypertable, Douglas Edric Stanley">

<p>One obvious but often overlooked candidate for further exploration of algorithmic writing would be the computer "desktop". In it's current laborious form, the desktop requires the slow and painful manipulation of endless morsels of discrete data packets in the form of files and folders. But even a cursory look at operations conducted via menus, drag and drop and selection would suggest that a potentially rich subset of programming switches can in fact be activated through fairly simple gestures. Adding on top of that the current trend towards multi-touch interfaces with their new and powerful vocabulary of actions, it would seem that desktop innovation is far from over. The risk is nevertheless that the algorithmic aspect of this environment, i.e. the modular and open-ended nature of what can be done with the desktop, will again be normalized around a specific set of limited actions that re-convert "users" from their newly-acquired skills of programming back into the actions mere data manipulators.</p>

</item>

<item keystroke="i" title="Software, Tool, Platform">

<p>One of the nagging problems of algorithmic writing-systems is the constant pull of software. Software is often considered the by-product of development, but this need not be the case. While programming itself is a dynamic practice, software is often packaged in a frozen (i.e. compiled) form, with the only window into this process taking place through endless cycles of "updates" (c.f. iPods/iTunes). Nevertheless, it is still a dynamic object in-and-of-itself, and can very easily become one of the ingredients of a larger platform, or even the nexus of an entirely new creation platform through use. One of the worst software periods for artists, and that we are only now collectively emerging from, was the re-invention of the development platform/compiler as an "authoring tool". The two major Macromedia platforms -- Flash and Director -- are good examples of this process and had previously locked over a decade's worth of artists and creative professionals into their workflow. It would appear that this stranglehold has finally begun to loosen, at least temporarily, thanks to projects such as Pure Data, Processing, or the fanciful platforms currently in development within the livecoding movement. Digital art exhibitions are very clearly no longer owned by software. In the world of video game development the field is still tightly locked down, although emerging platforms such as the Torque Engine are slowly making inroads into that field as well, as are parallel development and distributions channels via the Homebrew movement. And while each of these platforms depend for practical reasons on proprietary platforms, many of these same platforms are slowly opening up -- as in the case of Java --, or are simply being replaced by much more accessible Linux distributions such as Ubuntu. Additionally, the hacking movement has made it more and more difficult to put the genie back into the bottle, and have re-invented the definition of software around rights of use.</p>

</item>

<item keystroke="j" title="Assemblage">

<p>The hacker ethic and its new user-rights agenda opens a window into yet another form of algorithmic writing: assemblage. This aspect is all about the configuration itself, the network, the diagram, the machine observed within its environment. Assemblage is the invention of new algorithms that manifest themselves from within the connections of the machine, rather than the output of the machine, or the original design of the machine. In English we use the term "workflow" to describe these sorts of Rube Goldberg-inspired machines, and many protocols have been invented by and for artists to manipulate them, among them MIDI and its network variant, OSC. In the same way that analog synthesizers were all about the patch cords, the algorithm in assemblage machines are all about the ways in which inputs and outputs are modulated through routing.</p>

</item>

<item keystroke="k" title="Simulations">

<p>If we are to approach these machines not as representation machines but instead as simulation machines, we would be well served to begin constructing visual and conceptual interfaces for manipulating these simulations. The Game of Life is one of the more pleasant forms of this branch of algorithmic writing: as an immanent field of algorithmic activity intimately tied to its visualization, programs can be constructed and run concurrently. if we had to use a classical computer science terminology we could call it an "interpreted" language with real-time compilation of the "code". Another advantage of The Game of Life is its pedagogical aspect within the process of physicalization: the image constructed dynamically within the simulation is almost bit for bit identical to the physical switching making it possible, with the exception of the algorithm that actually manipulates the field (we do not have a visualization of this aspect). However, The Game of Life cannot be considered the fundamental structure of all simulation machines. This is one of the dangers of elegant simulations: we come to believe so convincingly in their representational appearance that we end up a Stephen Wolfram suggesting that the Universe itself might be nothing more than a cellular automata executed by a few simple lines of code. Nevertheless, cellular automata, neural networks, heuristic computing, and other "emergent" forms of computation are an essential part of manipulating abstractions. They are in fact probably the best candidates for replacing the recursive sophistication of text-based computer code.</p>

</item>

<item keystroke="l" title="Robotics">

<p>While physicalization is a concept tied to the historical inception of the first working computers, it could also be argued that a more tangible form of physicalization has always been at work in computing via robotics. While we do not have the time here to explore this aspect, we can point to the seminal work of Rosenblueth, Wiener, Bigelow on teleological machines as one of the significant steps of eternally tying up principles of feedback and environment both mechanically and biologically. Treating these principles (whatever the scientific validity) as our base epistemological material, the question shifts to

the nature of the robotic interfaces in the programmable process. Of all the forms of algorithmic writing discussed so far, it is notable that none have explicitly suggested a physical experience -- a difficult proposition for most artists. Robotics would be the answer to this problem, and will eventually approach the all-pervasive Graphical User Interface (GUI) as a valid form of algorithmic manipulation. The question remains how to build tangible objects that maintain both the openness and abstraction of the more electronic forms of programming. Tying robotics and physical interfaces to software is one answer and one that we have been living for the past several decades. But I would suggest another route, i.e. the use of robotics itself as software, with the physicality of the apparatus becoming the actual malleable material with the sophistication of a Game of Life.</p>

</item>

<item keystroke="m" title="Public Interfaces" image="synchronizer.png" imagetag="http://www.abstractmachine.net/syncrhonizer/">

<p>On the opposite side of the physical world, the so-called "cloud" can be considered another option for computer programming. Here the logic is that of the "public API": opening up public entrance and exit points for various external software, interfaces and platforms and allowing for access and appropriation through use. The principle qualifier for public interfaces as programmable machines lies in their recursive nature. This aspect is already visible in the infinite variety of RSS-feed mash-ups: we are no longer in a single-layer system of connected data; data is now combined into ever increasing stratifications of data conglomerates inside of data conglomerates. Once that resultant data is used for the further aggregation of new data, that data has effectively transformed itself into program code.</p>

</item>

<item keystroke="n" title="Visual Programming">

<p>Of the current generation of programming platforms, visual programming is probably the most attractive. The Patcher (a.k.a. Max, a.k.a. Pure Data), vvvv, and Eyes Web have proven themselves perfectly viable platforms for complex artistic installations and software. As an added bonus, these platforms tend to be run-time environments and allow for manipulation of the algorithm during its execution, much like The Game of Life. While these platforms have yet to solve the problems of scale and more abstract forms of recursivity (à la text-based programming), visual programming environments nevertheless offer an even more important aspect of algorithms: a bird's-eye visibility of the algorithm itself as a visible figure.</p>

</item>

<item keystroke="o" title="ASCII">

<p>With all of these past and future evolutions in mind, we might ask why then has text-based programming remained the dominant form of algorithmic writing when there are clearly more human-accessible

means available for programming? The first answer to this question would probably have something to do with the previously mentioned power of abstraction available when the programmer has in their arsenal directly: recursivity, pointers, classes, polymorphism, and so on, with the ability to scale these concepts in high volume. But I would like to suggest another reason for the predominance of text in programming, and that would be its historical relationship with the famous 7-bit character table known as ASCII. The importance of the ASCII table is not just its capacity to convert numerical values into alphanumerical values, but more importantly to communicate with the machine in a semantic form that brings closer the human and computer realms in a surprisingly efficient way. Talking in terms of code structures, the ASCII code allows us to string together digital values into one of the most important forms for computing: the array. The array is basically a series of data arranged linearly. Arrays are used in an amazing breath of computer expressions: music, images, communications, and here, text. Almost any activity inside of a computer can be contained within an array. The word processor I used to write this talk, is an array.</p>

</item>

<item keystroke="p" title="linearity vs. multithreading" image="americas_army.png" imagetag="America's Army, Eddo Stern">

<p>The trick of text-based programming grows from the fact that most computers are, like a traditional narrative, executed linearly even if the data and instructions that are placed within that line can come from any direction. So text is probably the easiest form of input, albeit merely as a "trick". Whoever has played Spacewar and its many descendants, knows that a computer keyboard can be used to control the computer without any linear structure whatsoever (c.f. "America's Army", Eddo Stern). This might also be one explanation for the obscure nature of programming and why code is so ugly. Computer code is ontologically quite removed from literature because ultimately is it just a string of perls. Thanks to the profound modularity of written language though, we have been able to tap into the extreme modularity of the computer. This however has its limits, as any programmer of a Playstation 3, with its multiple "cell" processors, has discovered the hard. If Sony wants to solve the economic woes foreshadowing their key platform, they might in fact want to hire music composers to program their machines, for these writers might better understand how to tame the complex harmonics of the machine.</p>

</item>

<item keystroke="q" title="Sandbox" image="littlebigplanet.png" imagetag="Little Big Planet, SONY">

<p>My final candidate today for the definition of computer programming platform are video games. While in the past I have had to argue through circuitous means that video games were a form of "deprogramming" computer code through use -- and were therefore a valid form of computer programming --, a new generation of "sandbox" games have appeared to fill in the gap and transform gaming into a more explicit form of "écriture algorithmique". In video games, the algorithm is the game, the game is the algorithm, simulation is its visualization. Gameplay executes code, embodies it, but also modulates it. Good gameplay is profoundly multithreaded, but it can also incorporate linear aspects, sign/token manipulation, visual

algorithms, physical interfaces (think "Wii"), and in new games such as Little Big Planet we even see tools of assemblage being coupled with public APIs for collective programming communities.</p>

</item>

<item keystroke="r" title="diagrams" subtitle="Douglas Edric Stanley" url="http://www.abstractmachine.net/diagrams/" image="diagram.png" imagetag="abstractmachine:diagram:transcode">

<p></p>

</item>

<item keystroke="s" title="^3" subtitle="Douglas Edric Stanley" url="http://www.abstractmachine.net/cubed/" image="cubed.png">

<p></p>

</item>

<item keystroke="t" title="checksum" subtitle="Douglas Edric Stanley">

<p></p>

</item>

</checksum>